CSE 5525 Homework #2: HMM and CRF for Named Entity Recognition

Deadline: **11:59PM on 10/06/2021**.

Academic Integrity

You can discuss the homework assignments with other students and study course materials together towards solutions. **However, all of the code and report you write must be your own!** If you do discuss or study together with others (except via posts/comments on Teams), please list their names at the top of your written submission. Unless there is academic misconduct detected, this won't be counted against your credit. (Plagiarism will be automatically detected on Carmen, e.g., by comparing your HW with all resources, as well as manually checked by TA.)

If the instructor or TA suspects that a student has committed academic misconduct in this course, they are obligated by university rules to report their suspicions to the Committee on Academic Misconduct. Please see more in Statements for CSE5525 on Carmen pages.

Goals

In this project, you'll first implement the Viterbi algorithm on a fixed model (an HMM) and then generalize that to forward-backward and implement learning and decoding for a feature-based CRF. This assignment serves two purposes: (1) You will be exposed to inference and learning for a simple structured model where exact inference is possible. (2) You will learn some of the engineering factors that need to be considered when implementing a model like this.

The expectation is that you understand CRFs and HMMs from scratch and can implement them based on the code provided to you. **You should not call existing HMM/CRF libraries in your solution to this assignment.**

Timeline & Credit

You will have around 4 weeks to work on this programming assignment. However, there are many events/deadlines in OCT such as Final Project Mid-term Report Due and Midterm. Late submissions will NOT be accepted. So, start early and plan ahead! We currently use a 100-point scale for this homework (Part 1: 30 points; part 2: 50 points; 1-2 page report: 20 points), but it will take 20% of your final grade.

Questions?

Please create a post on MS teams to get timely help from other students, the TA, and the instructor. Remember that participation takes 5% of your final grade. You can show your participation by actively answering others' questions or participating the discussion!

1 Background and Dataset

Named entity recognition is the task of identifying references to named entities of certain types in text. We use data presented in the CoNLL 2003 Shared Task (Tjong Kim Sang and De Meulder, 2003; [1]). An

example of a data instance (e.g., "Singapore Refining Company expected to shut CDU 3.") is given below:

```
Singapore NNP I-NP B-ORG
Refining NNP I-NP I-ORG
Company NNP I-NP I-ORG
expected VBD I-VP O
to TO I-VP O
shut VB I-VP O
CDU NNP I-NP B-ORG
3 CD I-NP I-ORG
. . O O
```

There are **four columns** here: the word, the POS tag, the chunk bit (a form of shallow parsing—you can ignore this), and **the column containing the NER tag**. NER labels are given in a BIO tag scheme: beginning, inside, outside. In the example above, two named entities are present: `Singapore Refining Company` and `CDU 3`. `O` tags denote text not part of a named entity. `B` tags indicate the start of a named entity, and `I` tags indicate the continuation of the previous named entity. Both `B` and `I` tags are hyphenated and contain a type after the hyphen, which in this dataset is one of `PER`, `ORG`, `LOC`, or `MISC`. One `B` tag can immediately follow another `B` tag in the case where a one-word entity is followed immediately by another entity. However, note that an `I` tag can only follow an `I` tag or `B` tag of the same type.

An NER system's job is to predict the NER chunks of an unseen sentence, i.e., predict the last column given the others. Output is typically evaluated according to chunk-level F-measure.[1] To evaluate a single sentence, let C denote the predicted set of labeled chunks represented by a tuple of (label, start index, end index) and let C^* denote the gold set of chunks. We compute precision, recall, and F_1 as follows:

$$\text{Precision} = \frac{|C \cap C^*|}{|C|}; \text{ Recall} = \frac{|C \cap C^*|}{|C^*|}; \text{ F}_1 = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

The gold labeled chunks from the example above are (`ORG`, 0, 3) and (`ORG`, 6, 8) (i.e., $C^* = \{(\texttt{ORG}, 0, 3), (\texttt{ORG}, 6, 8)\}$) using 0-based indexing and semi-inclusive notation for intervals.

To generalize to corpus-level evaluation, the numerators and denominators of precision and recall are aggregated across the corpus. State-of-the-art systems can get above 90 F_1 on this dataset; you should be aiming to get close to this and build systems that can get in at least the mid-80s.

1.1 Data

Data `eng.train` is the training set, `eng.testa` is the standard NER development set, and `eng.testb.blind` is a blind test set you will submit results on. The `deu*` files are German data, which you're **NOT** required to do anything with in this assignment (but, feel free to consider using them for your final project).

1.2 Getting started

Download the data and code. You will need Python 3 and numpy.[2] Try running:

```
python ner.py
```

This will run a very bad NER system, which simply outputs the most common tag for every token, or `O` if it hasn't been seen before. The system will print a bunch of warnings on the dev set—this baseline doesn't even produce consistent tag sequences.

1.3 Framework code

You have been given the framework code to start with:

`ner.py`: Contains the implementation of `BadNerModel` and the main function which reads the data, trains the appropriate model, and evaluates it on the test set.

[1] Tag-level accuracy isn't used because of the prevalence of the `O` class—always predicting O would give extremely high accuracy!

[2] If you don't have numpy, see `https://www.scipy.org/install.html`. You may try installing with anaconda.

`nerdata.py`: Utilities for reading NER data, evaluation code, and functions for converting from `BIO` to chunk representation and back. The main abstraction to pay attention to here is `LabeledSentence`, which contains a sequence of `Token` objects (wrappers around words, POS tags, and chunk information) and a set of `Chunk` objects representing a labeling. Gold examples are `LabeledSentence` and this is also what your system will return as predictions.

`utils.py`: `Indexer` and `Counter` are as HW#1, with `Indexer` additionally being useful for mapping between labels and indices in dynamic programming. A `Beam` data structure is also provided, but you won't need that in this project.

`optimizers.py`: Three optimizer classes implementing SGD, unregularized Adagrad, and L1-regularized Adagrad. These wrap the weight vector, exposing `access` and `score` methods to use it, and are updated by `apply_gradient_update`, which takes as input a `Counter` of feature values for this gradient as well as the size of the batch the gradient was computed on.

`models.py`: You should feel free to modify anything in this file as you need, but the scaffolding will likely serve you well. We will describe the code here in more detail in the following sections.

Next, try running:
`python ner.py --model HMM`
This will crash with an error message. You have to implement Viterbi decoding (Part 1 below) to make the HMM work.

2 What You Need to Do

2.1 Part 1: Viterbi Decoding (30 points)

Go to `models.py`. The `train_hmm_model` estimates initial state, transition, and emission probabilities from the labeled data and returns a new `HmmNerModel` with these probabilities. Your task is to implement Viterbi decoding in this model, so it can return `LabeledSentence` predictions in `decode`.

We've provided an abstraction for you in `ProbabilisticSequenceScorer`. This abstraction is meant to help you build inference code that can work for both generative and probabilistic scoring as well as feature-based scoring. `score_init` scores the initial HMM state, `score_transition` scores an HMM state transition, and `score_emission` scores the HMM emission. All of these are implemented as log probabilities. Note that this abstraction operates in terms of indexed tags, where the indices have been produced by `tag_indexer`. This allows you to score tags directly in the dynamic programming state space without converting back and forth to strings all the time.

You should implement the Viterbi algorithm with scores that come from log probabilities to find the highest log-probability path.

$$P(y_1, ..., y_n | x_1, ..., x_n) \propto P(y_1, ..., y_n, x_1, ..., x_n) = P(y_1)\Big[\prod_{i=2}^{n} P(y_i|y_{i-1})\Big]\Big[\prod_{i=1}^{n} P(x_i|y_i)\Big]$$

$$\text{argmax}_{y_1,...,y_n} P(y_1, ..., y_n | x_1, ..., x_n) = \text{argmax}_{y_1,...,y_n}\Big\{\log P(y_1) + \Big[\sum_{i=2}^{n} \log P(y_i|y_{i-1})\Big] + \Big[\log \sum_{i=1}^{n} P(x_i|y_i)\Big]\Big\}$$

If you are not sure about the interface to the code, take a look at `BadNerModel` decoding and use that as a template.

Grading expectation. Viterbi decoding for the HMM can get above 75 F_1 on the development set. You could take this as a reference and debug your implementation if its performance is substantially lower. Note that a very low performance here might imply the incorrectness of your code, which can result in partial credit for this part as well as cause trouble for implementing Part 2 correctly.

Implementation tips. You have a lot of freedom to determine the implementation details, but here are some implementation tips:

1. Python data structures like lists and dictionaries can be pretty inefficient. Consider using numpy arrays in dynamic programs.

2. Once you run your dynamic program, you still need to extract the best answer. Typically this is done by either storing a backpointer for each cell to know how that cell's value was derived or by using a backward pass over the chart to reconstruct the sequence.

2.2 Part 2: CRF Training (50 points)

In the CRF training phase, you will implement learning and inference for a CRF sequence tagger with a fixed feature set. We provide a simple CRF feature set with emission features only. While you'll need to impose some kind of sequential constraints in the model, transition features are often slow to learn: you should be able to get good performance by constraining the model to only produce valid BIO sequences (prohibiting a transition to I-X from anything except I-X and B-X).

We provide a code skeleton in `CrfNerModel` and `train_crf_model`. The latter calls feature extraction in `extract_emission_features` and builds a cache of features for each example — you can feel free to use this cache or not.

You should take the following steps to implement your CRF:

1. Generalize your Viterbi code to forward-backward.
2. Extend your forward-backward code to use a scorer based on features—perhaps you might write a `FeatureBasedSequenceScorer` with an interface similar to `ProbabilisticSequenceScorer` for this purpose.
3. Compute the stochastic gradient of the feature vector for a sentence.
4. Use the stochastic gradient in a learning loop to learn feature weights for the NER system.

Grading expectation. You should be able to get a score of at least 85 F1 on the development set. Assignments falling short of this will be judged based on completeness and awarded partial or full credit accordingly: If you get at least 75-85, for example, we can see that your implementation may be mostly working and you will receive substantial credit, and please conduct error analysis to analyze your implementation. [One reference implementation gets 88.2 F1 using the given emission features and unregularized Adagrad as the optimizer—see if you can beat that!]

Implementation tips. You have a lot of freedom to determine the implementation details, but here are some implementation tips:

- Make sure that your probabilities from forward-backward normalize correctly! You should be able to sum the forward x backward chart values at each sentence index and get the same value (the normalizing constant). You can check your forward-backward implementation in the HMM model if that's useful.
- When implementing forward-backward, you'll probably want to do so in log space rather than real space. (+, x) in real space translates to (log-sum-exp, +) in log space. Use `numpy.logaddexp`.
- Remember that the NER tag definition has hard constraints in it (only B-X or I-X can transition to I-X), so be sure to build these into your inference procedure. You can also incorporate features on tag pairs, but this is substantially more challenging and not necessary to get good performance.
- If your code is too slow, try (a) making use of the feature cache to reduce computation and (b) exploiting sparsity in the gradients (`Counter` is a good class for maintaining sparse maps). Run your code with `python -m cProfile ner.py` to print a profile and help identify bottlenecks.
- Implement things in baby steps! First make sure that your marginal probabilities look correct on a single example. Then make sure that your optimizer can fit a very small training set (10 examples); you might want to write a small amount of extra code to compute log-likelihood and check that this goes up, along with train accuracy. Then work on scaling things up to more data and optimizing for development performance.

3 What You Should Submit and Our Expectation

You should submit the following to Carmen:

- Your code as a .zip or .tgz file
- Your CRF's output on the `testb` blind test set in CoNLL format.

- A report of around 1-2 pages in PDF, not including references, though you aren't expected to reference many papers. Your report should list your collaborators (i.e., who you discussed HW#2 with, if any; see the beginning of this pdf), **briefly** state what you are doing, describe relevant details of your implementation, present results on the development set from CRF, and briefly discuss a few error cases and how the system could be further improved.

Grading Expectation. Your assignment is judged roughly 80% based on your implementation (30% for part 1 and 50% for Part 2) and 20% on the report. **Even if you cannot make your implementation work, describe your effort.**

3.1 Acknowledgment

This homework assignment is largely adapted from NLP courses by Dr. Greg Durrett at UT Austin.

References

[1] Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.